JZarr

Apr 23, 2022

Contents

JZarr is a Java library providing an implementation of chunked, compressed, N-dimensional arrays close to the Python zarr package.

CHAPTER 1

Highlights

- Create N-dimensional arrays with java primitive data types. At the moment boolean and char type are not supported.
- Chunk arrays along any dimension.
- Compress the chunks by using one of the *compressors*
- Store arrays in memory, on disk, (Future plans: inside a Zip file, on S3, . . .)
- Read an array concurrently from multiple threads or processes.
- Write an array concurrently from multiple threads or processes.
- Organize arrays into hierarchies via groups.

CHAPTER 2

Status

JZarr is in the very beginning phase. Feedback and bug reports are very welcome. Please get in touch via the GitHub issue tracker.

CHAPTER 3

Requirements

3.1 Java

JZarr needs Java 8 or higher.

3.2 blosc

- This API also offers blosc compression. **:raw-html:‘
‘** To use this compression, a compiled c-blosc distributed library must be available on the operating system.
- If such a library is not available ... The C sourcecode and instructions to build the library can be found at https://github.com/Blosc/c-blosc.
- If you want to use the JZarr API and the integrated blosc compression, you have to start the Java Virtual Machine with the following VM parameter:

```
-Djna.library.path=<path which contains the compiled c-blosc library>
```

CHAPTER 4

Maven Dependency

To use JZarr in your project integrate the following lines into your maven pom.xml:

```
<dependency>
   <groupId>com.bc.zarr</groupId>
   <artifactId>jzarr</artifactId>
   <version>0.3.5</version>
</dependency>

<repositories>
   <repository>
      <id>bc-nexus-repo</id>
      <name>Brockmann-Consult Public Maven Repository</name>
      <url>https://nexus.senbox.net/nexus/content/groups/public/</url>
   </repository>
</repositories>
```

It is planned to deploy the JZarr artifacts to the maven central repository too.

CHAPTER 5

Contents

5.1 JZarr documentation

JZarr is a Java library providing an implementation of chunked, compressed, N-dimensional arrays close to the Python zarr package.

5.1.1 Highlights

- Create N-dimensional arrays with java primitive data types. At the moment boolean and char type are not supported.
- Chunk arrays along any dimension.
- Compress the chunks by using one of the *compressors*
- Store arrays in memory, on disk, (Future plans: inside a Zip file, on S3, . . .)
- Read an array concurrently from multiple threads or processes.
- Write an array concurrently from multiple threads or processes.
- Organize arrays into hierarchies via groups.

5.1.2 Status

JZarr is in the very beginning phase. Feedback and bug reports are very welcome. Please get in touch via the GitHub issue tracker.

5.1.3 Requirements

Java

JZarr needs Java 8 or higher.

blosc

- This API also offers blosc compression. To use this compression, a compiled c-blosc distributed library must be available on the operating system.
- If such a library is not available ... The C sourcecode and instructions to build the library can be found at https://github.com/Blosc/c-blosc.
- If you want to use the JZarr API and the integrated blosc compression, you have to start the Java Virtual Machine with the following VM parameter:

```
-Djna.library.path=<path which contains the compiled c-blosc library>
```

5.1.4 Maven Dependency

To use JZarr in your project integrate the following lines into your maven pom.xml:

```
<dependency>
   <groupId>com.bc.zarr</groupId>
   <artifactId>jzarr</artifactId>
   <version>0.3.5</version>
</dependency>

<repositories>
   <repository>
      <id>bc-nexus-repo</id>
      <name>Brockmann-Consult Public Maven Repository</name>
      <url>https://nexus.senbox.net/nexus/content/groups/public/</url>
   </repository>
</repositories>
```

It is planned to deploy the JZarr artifacts to the maven central repository too.

5.2 Tutorial

JZarr provides classes and functions to handle N-dimensional arrays data whose data can be divided into chunks and each chunk can be compressed.

5.2.1 General Information

In the JZarr API, data inputs and outputs are allways one-dimensional arrays of primitive Java types `double`, `float`, `long`, `int`, `short`, `byte`. Users have to specify the N-dimensionality of the data by a shape parameter requested by many of the JZarr API operations.

To read or write data portions to or from the array, a shape describing the portion and an offset is needed. The zarr array offsets are zero-based (`0`).

For Example:
If you need to write data to the upper left corner of a 2 dimensional zarr array you have to use an offset of `new int[]{0, 0}`.

Note: All data persisted using this API can be read in with the Python zarr API without limmitations.

If you are already familiar with the Python zarr package then JZarr provide similar functionality, but without NumPy array behavior.

If you need array objects which behave almost like NumPy arrays you can wrap the data using ND4J INDArray from deeplearning4j.org. You can find examples in the data writing and reading examples below.

Alternatively you can use `ucar.ma2.Array` from netcdf-java Common Data Model to wrap the data.

5.2.2 Creating an array

JZarr has several functions for creating arrays. For example:

Listing 1: example 1 from Tutorial_rtd.java

```
ZarrArray jZarray = ZarrArray.create(new ArrayParams()
        .shape(10000, 10000)
        .chunks(1000, 1000)
        .dataType(DataType.i4)
);
```

A `System.out.println(array);` then creates the following output

Listing 2: output

```
com.bc.zarr.ZarrArray{'/' shape=[10000, 10000], chunks=[1000, 1000], dataType=i4,␣
↪fillValue=0, compressor=blosc/cname=lz4/clevel=5/blocksize=0/shuffle=1,␣
↪store=InMemoryStore, byteOrder=BIG_ENDIAN}
```

The code above creates a 2-dimensional array of 32-bit integers with 10000 rows and 10000 columns, divided into chunks where each chunk has 1000 rows and 1000 columns (and so there will be 100 chunks in total).

For a complete list of array creation routines see the array creation module documentation.

5.2.3 Writing and reading data

This example shows how to write and read a region to an array.

Creates an array with size [5 rows, 7 columns], with data type `int` and with a fill value of `-9999`.

Listing 3: example 2 from Tutorial_rtd.java

```
ZarrArray array = ZarrArray.create(new ArrayParams()
        .shape(5, 7)
        .dataType(DataType.i4) // integer data type
        .fillValue(-9999)
);
```

Prepare the data which should be written to the array with a shape of [3, 5] and an offset of [2, 0].

```
// define the data which should be written
int[] data = {
        11, 12, 13, 14, 15,
        21, 22, 23, 24, 25,
```

(continues on next page)

(continued from previous page)

```
        31, 32, 33, 34, 35
};
int[] shape = {3, 5}; // the actual N-D shape of the data
int[] offset = {2, 0}; // and the offset into the original array
```

Write the prepared data.

```
array.write(data, shape, offset);
```

Read the entire data from the array.

```
int[] entireData = (int[]) array.read();
```

Print out the data read.

```
OutputHelper.Writer writer = out -> {
    DataBuffer buffer = Nd4j.createBuffer(entireData);
    out.println(Nd4j.create(buffer).reshape('c', array.getShape()));
};
```

Creates the following output

Listing 4: output

```
[[     -9999,     -9999,     -9999,     -9999,     -9999,     -9999,     -9999],
 [     -9999,     -9999,     -9999,     -9999,     -9999,     -9999,     -9999],
 [        11,        12,        13,        14,        15,     -9999,     -9999],
 [        21,        22,        23,        24,        25,     -9999,     -9999],
 [        31,        32,        33,        34,        35,     -9999,     -9999]]
```

The output displays that the previously written data (written with an offset of [2, 0]) is where we expect it to be. Other positions contain the previously defined fill value `-9999`.

Note: Nd4j is not part of the JZarr library. It is only used in this showcase to demonstrate how the data can be used.

5.2.4 Persistent arrays

In the examples above, the data of the arrays (chunked) were only stored in RAM. JZarr arrays can also be stored on a file system, which enables persistence of data between sessions.

Listing 5: example 3 from Tutorial_rtd.java

```
String path = "docs/examples/output/example_3.zarr";
ZarrArray createdArray = ZarrArray.create(path, new ArrayParams()
        .shape(1000, 1000)
        .chunks(250, 250)
        .dataType(DataType.i4)
        .fillValue(-9999)
);
```

The array above will store its configuration metadata (zarr header `.zarray`) and all compressed chunk data in a directory called 'docs/examples/output/example_3.zarr' relative to the current working directory.

The created zarr header file .zarray written in JSON format.

```
{
  "chunks" : [ 250, 250 ],
  "compressor" : {
    "clevel" : 5,
    "blocksize" : 0,
    "shuffle" : 1,
    "cname" : "lz4",
    "id" : "blosc"
  },
  "dtype" : ">i4",
  "fill_value" : -9999,
  "filters" : null,
  "order" : "C",
  "shape" : [ 1000, 1000 ],
  "dimension_separator" : ".",
  "zarr_format" : 2
}
```

Write some data to the created persistent array.

```
int value = 42;
int[] withShape = {3, 4};
int[] toPosition = {21, 22};

createdArray.write(value, withShape, toPosition);
```

Note: There is no need to close an array. Data are automatically flushed to disk, and files are automatically closed whenever an array is modified.

Then we can reopen the array and read the data

```
ZarrArray reopenedArray = ZarrArray.open("docs/examples/output/example_3.zarr");
int[] shape = {5, 6};
int[] fromPosition = {20, 21};
int[] data = (int[]) reopenedArray.read(shape, fromPosition);
```

Which now looks like:

Listing 6: output

```
[[     -9999,     -9999,     -9999,     -9999,     -9999,     -9999],
 [     -9999,        42,        42,        42,        42,     -9999],
 [     -9999,        42,        42,        42,        42,     -9999],
 [     -9999,        42,        42,        42,        42,     -9999],
 [     -9999,     -9999,     -9999,     -9999,     -9999,     -9999]]
```

5.2.5 Resizing and appending

Currently not implemented.

5.2.6 Compressors

A number of different compressors can be used with JZarr. Different compressors can be created and parameterized with the CompressorFactory.

blosc `default compressor`

Default values are:

```
cname = "lz4"
clevel: 5
blocksize: 0
shuffle: 1
```

The meaning of this values you can find at: Official c-blosc API documentation

Valid values are:

```
cname ... "zstd", "blosclz", "lz4", "lz4hc", "zlib"
clevel ... clevel parameter must be between 0 and 9
blocksize ... see https://github.com/Blosc/c-blosc/blob/master/blosc/blosc.h#L202
shuffle   ... -1 (AUTOSHUFFLE) / 0 (NOSHUFFLE) / 1 (BYTESHUFFLE=default) / 2␣
↪(BITSHUFFLE)
```

zlib

Default values are:

```
level: 1
```

Valid values are:

```
level ... level parameter must be between 0 and 9
```

NULL

The *null* compressor implements the Compressor interface, but does not apply any compression algorithm. If you want to easily verify exactly what data is being written while developing an application that uses JZarr, it can be helpful to use this compressor. The written data then can be easily verified with a hex editor.

Examples

Listing 7: example 4 from Tutorial_rtd.java

```
// Currently available compressors

// zlib compressor
// ===============
// creates a zlib compressor with standard compression level 1
CompressorFactory.create("zlib");

// creates a zlib compressor with compression level 8 ... valid values 0 .. 9
CompressorFactory.create("zlib", "level", 8);

// blosc compressor
// ================
// All permutations of cname, clevel, shuffle and blocksize are allowed

// creates a blosc compressor with standard values
CompressorFactory.create("blosc");

// creates a blosc compressor with lz4hc algorithm with level 7
CompressorFactory.create("blosc", "cname", "lz4hc", "clevel", 7);

// creates a blosc compressor with byteshuffle
CompressorFactory.create("blosc", "shuffle", 1);

// null compressor
// ===============
// Means no compression e.g. for analysis purposes
// In this case values are written without compression as raw binaries
Compressor compNull = CompressorFactory.create("null");

ZarrArray jZarray = ZarrArray.create(new ArrayParams()
        .shape(222, 333, 44)  // one or more dimensions
        .compressor(compNull)  // if you need uncompressed chunks e.g. for analysis␣
↪purposes
);
```

Note: In this very beginning phase we only implemented the **"blosc"**, the **"zlib"** and a **"null"** compressor. If no compressor is specified at array creation time, a **"blosc"** compressor with default values is used. More compressors will be implemented in the future.

Additionally, in the future, developers should be able to register their own Compressors in the CompressorFactory. A compressor must extend the abstract Compressor class.

5.2.7 Filters

Currently not implemented.

5.2.8 Groups

JZarr supports hierarchical organization of arrays via groups. As with arrays, groups can be stored in memory, on disk, or via other storage systems that support a similar interface.

To create a group, use the suitable static ZarrGruup.create() method.

In the following example you can see:

- how to create a group
- how to create sub groups
- how to create arrays within a group
- how to open existing groups
- how to open existing subgroups
- how to open existing arrays within a group or subgroup

Listing 8: example 5 from Tutorial_rtd.java

```
String path = "docs/examples/output/example_5.zarr";

// persist a group with sub groups and an array
ZarrGroup root = ZarrGroup.create(path);
ZarrGroup foo = root.createSubGroup("foo");
ZarrGroup bar = foo.createSubGroup("bar");
ZarrArray array = bar.createArray("baz", new ArrayParams()
        .shape(1000, 1000).chunks(100, 100).dataType(DataType.i4)
);

// reopen the root group, nested groups and array
final ZarrGroup rootReopened = ZarrGroup.open(path);
final Iterator<String> groupKeyIter = rootReopened.getGroupKeys().iterator();
final ZarrGroup sub1 = rootReopened.openSubGroup(groupKeyIter.next());
final ZarrGroup sub2 = rootReopened.openSubGroup(groupKeyIter.next());
final ZarrArray arrayReopened = rootReopened.openArray(rootReopened.getArrayKeys().
↪iterator().next());
```

The following code snipped

```
    printStream.println(sub1);
    printStream.println(sub2);
    printStream.println(arrayReopened);
```

then creates the following output

```
com.bc.zarr.ZarrGroup{'/foo'}
com.bc.zarr.ZarrGroup{'/foo/bar'}
com.bc.zarr.ZarrArray{'/foo/bar/baz' shape=[1000, 1000], chunks=[100, 100],␣
↪dataType=i4, fillValue=0.0, compressor=blosc/cname=lz4/clevel=5/blocksize=0/
↪shuffle=1, store=FileSystemStore, byteOrder=BIG_ENDIAN}
```

5.2.9 User attributes

JZarr arrays and groups support custom key/value attributes, which can be useful for storing application-specific metadata. For example:

Listing 9: example 6 from Tutorial_rtd.java

```
Map<String, Object> attrs;
Map<String, Object> attributes;
ZarrGroup group;
ZarrArray array;

attrs = new HashMap<>();
attrs.put("baz", 42);
attrs.put("qux", new int[]{1, 4, 7, 12});

// store user attributes at group creation time
group = ZarrGroup.create(new InMemoryStore(), attrs);

// store user attributes at array creation time
array = ZarrArray.create(new ArrayParams().shape(10, 10), attrs);

// also at array creation time within a group
array = group.createArray("name", new ArrayParams().shape(10, 10), attrs);

// store or restore user attributes after creation time
group.writeAttributes(attrs);
array.writeAttributes(attrs);

// get attributes from group
attrs = group.getAttributes();
// similar from array
attributes = array.getAttributes();
```

You can easily print out the attributes content using `System.out.println(ZarrUtils.toJson(attributes, true));`

Listing 10: output

```
{
  "qux" : [ 1, 4, 7, 12 ],
  "baz" : 42
}
```

Note: If you take user attributes from a group or an array modifications (put, replace or remove) on the attributes are not automatically stored. The `writeAttributes()` from ZarrGroup or ZarrArray must be used to restore the changed attributes.

Internally JZarr uses JSON to store array attributes, so attribute values must be JSON serializable.

5.2.10 Partly reading writing

JZarr ZarrArrays enable a subset of data items to be extracted or updated in an array without loading the entire array into memory.

Listing 11: example 7 from Tutorial_rtd.java

```
// create an array
int height = 10;
```

(continues on next page)

(continued from previous page)

```
int width = 6;
int[] arrayShape = {height, width};
ZarrArray arr = ZarrArray.create(new ArrayParams()
        .shape(arrayShape).dataType(DataType.i2));

// write a data part (value 33) to the offset position with the given shape
int value = 33;
int[] shape = {6, 4};   // the shape for the data to be written
int[] offset = {2, 1};  // y direction offset: 2   x direction offset: 1
arr.write(value, shape, offset);

// read entire array data
short[] data = (short[]) arr.read();

// wrap the data and create output
createOutput(out -> {
    DataBuffer buffer = Nd4j.createBuffer(arrayShape, org.nd4j.linalg.api.buffer.
↪DataType.SHORT);
    buffer.setData(data);
    out.println(Nd4j.create(buffer).reshape('c', arrayShape));
});
```

The output shows the data is written with the given shape to the offset position.

Listing 12: output

```
[[         0,         0,         0,         0,         0,         0],
 [         0,         0,         0,         0,         0,         0],
 [         0,   33.0000,   33.0000,   33.0000,   33.0000,         0],
 [         0,   33.0000,   33.0000,   33.0000,   33.0000,         0],
 [         0,   33.0000,   33.0000,   33.0000,   33.0000,         0],
 [         0,   33.0000,   33.0000,   33.0000,   33.0000,         0],
 [         0,   33.0000,   33.0000,   33.0000,   33.0000,         0],
 [         0,   33.0000,   33.0000,   33.0000,   33.0000,         0],
 [         0,         0,         0,         0,         0,         0],
 [         0,         0,         0,         0,         0,         0]]
```

Partly read and write is also used in examples above. See *Persistent arrays*

5.2.11 Chunk size an shape

In general, chunks of at least 1 megabyte (1M) uncompressed size seem to provide good performance, at least when using compression too.

The optimal chunk shape will depend on how you want to access the data. E.g., for a 2-dimensional array, if you only ever take slices along the first dimension, then chunk across the second dimenson. If you know you want to chunk across an entire dimension you can use `0` or `negative value` within the chunks argument, e.g.:

Listing 13: example 8 from Tutorial_rtd.java

```
ZarrArray zarray = ZarrArray.create(new ArrayParams()
        .shape(8888, 7777).chunks(100, 0)
);
int[] chunks = zarray.getChunks();
```

The output shows the automatically replaced `0` with full size of the first dimension.

Listing 14: output

```
[100, 7777]
```

Alternatively, if you only ever take slices along the second dimension, then chunk across the first dimension, e.g.:

Listing 15: example 9 from Tutorial_rtd.java

```
ZarrArray zarray = ZarrArray.create(new ArrayParams()
        .shape(8888, 7777).chunks(0, 100)
);
int[] chunks = zarray.getChunks();
```

The output shows the automatically replaced `0` with full size of the second dimension.

Listing 16: output

```
[8888, 100]
```

If you require reasonable performance for both access patterns then you need to find a compromise, e.g.:

Listing 17: example 10 from Tutorial_rtd.java

```
ZarrArray zarray = ZarrArray.create(new ArrayParams()
        .shape(10000, 10000).chunks(1000, 1000)
);


ate static void example_11() throws IOException, JZarrException {
ZarrArray zarray;
zarray = ZarrArray.create(new ArrayParams()
        .shape(6200, 7500).chunked(true)
);
int[] chunks1 = zarray.getChunks();

// array creation without chunked(true) leads to the same result, because true is the
// default value for the parameter chunked
zarray = ZarrArray.create(new ArrayParams()
        .shape(6200, 7500)
);
int[] chunks2 = zarray.getChunks();
```

If you are feeling lazy, you can let JZarr guess a chunk shape for your data by providing `chunked(True)` E.g.:

Listing 18: example 11 from Tutorial_rtd.java

```
ZarrArray zarray;
zarray = ZarrArray.create(new ArrayParams()
        .shape(6200, 7500).chunked(true)
);
int[] chunks1 = zarray.getChunks();

// array creation without chunked(true) leads to the same result, because true is the
// default value for the parameter chunked
zarray = ZarrArray.create(new ArrayParams()
        .shape(6200, 7500)
);
```

(continues on next page)

(continued from previous page)

```
int[] chunks2 = zarray.getChunks();
```

Listing 19: output

```
chunks1 = [477, 500]
chunks2 = [477, 500]
```

Note: The algorithm for guessing a chunk shape is based on simple heuristics and may be far from optimal.

If you know you are always going to be loading the entire array into memory, you can turn off chunks by providing `chunked(false)`, in which case there will be one single chunk for the array:

Listing 20: example 12 from Tutorial_rtd.java

```
ZarrArray zarray = ZarrArray.create(new ArrayParams()
        .shape(6200, 7500).chunked(false)
);
int[] chunks = zarray.getChunks();
```

Listing 21: output

```
chunks = [6200, 7500]
```

5.2.12 Chunk memory layout

The order of bytes within chunks of an array can be changed via the `byteOrder()` parameter. To use either `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN` layout. E.g.:

Listing 22: example 13 from Tutorial_rtd.java

```
ZarrArray zarray = ZarrArray.create(new ArrayParams()
        .shape(6200, 7500)
        .byteOrder(ByteOrder.BIG_ENDIAN)
);
```

These two layouts may provide different compression ratios, depending on the correlation structure within the data. Changing the order of bytes within chunks of an array may improve the compression ratio, depending on the structure of the data, the compression algorithm/level is used.

Note: If Byte order is not set, the default Byte order `ByteOrder.BIG_ENDIAN` will be used.

5.2.13 Parallel computing and synchronisation

Basically zarr arrays have been designed for use as the source or sink for data in parallel computations. By data source we mean that multiple concurrent read operations may occur. By data sink we mean that multiple concurrent write operations may occur, with each writer updating a different array.

Additionally JZarr is thread save in cases if multiple write operations occur on the same array on the same chunk concurrently from within different threads. This works without data loss.

Currently JZarr is not process save. This means, that if multiple write operations occur on the same array on the same chunk concurrently from within different processes, this can lead to data loss. Process synchronizing will be implemented in the future too.

When using a JZarr array as a data sink, some synchronization (locking) may be required to avoid data loss, depending on how data are being updated.

If each worker in a parallel computation is writing to a separate region of the array, and if region boundaries are perfectly aligned with chunk boundaries, then no synchronization is required. However, if region and chunk boundaries are not perfectly aligned, then synchronization is required to avoid two workers attempting to modify the same chunk at the same time, which could result in data loss.

To give a simple example, consider a 1-dimensional array of length 30, z, divided into three chunks of 10 elements each. If three worker processes are running and each attempts to write to a 10 element region (i.e., offset {0}, offset {10} and offset {20}) then each worker will be writing to a separate chunk and no synchronization is required.

Listing 23: example 14 from Tutorial_rtd.java

```
ZarrArray z = ZarrArray.create(new ArrayParams().shape(30).chunks(10).
↪dataType(DataType.i4));

int[] writeDataShape = {10};

int[] data1 = {10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
int[] data2 = {20, 21, 22, 23, 24, 25, 26, 27, 28, 29};
int[] data3 = {30, 31, 32, 33, 34, 35, 36, 37, 38, 39};

int[] offset1 = {0};
int[] offset2 = {10};
int[] offset3 = {20};

z.write(data1, writeDataShape, offset1);
z.write(data2, writeDataShape, offset2);
z.write(data3, writeDataShape, offset3);

int[] data = (int[]) z.read();
```

Listing 24: output

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,␣
↪31, 32, 33, 34, 35, 36, 37, 38, 39]
```

5.3 How to use JZarr with AWS S3

In general JZarr can work with `java.nio.file.Path` objects. So if someone extends the abstract `java.nio.file.FileSystem` (see FileSystem) to connect an AWS S3 bucket this can be used to read from and write directly to such buckets.

In our example we use the Amazon-S3-FileSystem-NIO2 library which is forked several times by other implementors.

If you want to try the following example, add this maven dependency to your pom:

```
<dependency>
    <groupId>org.lasersonlab</groupId>
    <artifactId>s3fs</artifactId>
```

(continues on next page)

(continued from previous page)

```
    <version>2.2.3</version>
</dependency>
```

Below you can see code snippets for **connecting** with, **writing** to and **reading** from an s3 bucket. You can find the entire example code here: S3Array_nio.java

5.3.1 connect an s3 bucket

Fill in your credentials.

Listing 25: code example for connecting the s3 bucket

```
String s3AccessKey = "<your access key>";
String s3SecretKey = "<your secret key>";
String s3Server = "s3.eu-central-1.amazonaws.com"; // example server name
String s3BucketName = "bucket-abcd"; // example bucket name

URI uri = URI.create(MessageFormat.format("s3://{0}:{1}@{2}", s3AccessKey,␣
↪s3SecretKey, s3Server));
FileSystem s3fs = FileSystems.newFileSystem(uri, null);
Path bucketPath = s3fs.getPath("/" + s3BucketName);
```

5.3.2 write to an s3 bucket

In this example data will be written without compression. So you can easily check the chunk file content e.g. with an hex file viewer.

Listing 26: code example writing to an s3 bucket

```
Path groupPath = bucketPath.resolve("GroupName.zarr");
ZarrGroup group = ZarrGroup.create(groupPath);
ZarrArray array = group.createArray("AnArray", new ArrayParams()
        .shape(4, 8).chunks(2, 4).dataType(DataType.i1).compressor(CompressorFactory.
↪nullCompressor));
byte[] data = {
        11, 12, 13, 14, 15, 16, 17, 18,
        21, 22, 23, 24, 25, 26, 27, 28,
        31, 32, 33, 34, 35, 36, 37, 38,
        41, 42, 43, 44, 45, 46, 47, 48
};
int[] shape = {4, 8};
int[] offset = {0, 0};
array.write(data, shape, offset);
```

5.3.3 read from an s3 bucket

Listing 27: code example reading from an s3 bucket

```
Path groupPath = bucketPath.resolve("GroupName.zarr");
final ZarrGroup group = ZarrGroup.open(groupPath);
ZarrArray array = group.openArray("AnArray");
```

(continues on next page)

(continued from previous page)

```
byte[] bytes = (byte[]) array.read();
System.out.println(Arrays.toString(bytes));
```

The System.out should produce the following output:

```
[11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 24, 25, 26, 27, 28, 31, 32, 33, 34, 35,␣
↪36, 37, 38, 41, 42, 43, 44, 45, 46, 47, 48]
```

5.4 License

```
MIT License

Copyright (c) 2020 Brockmann Consult GmbH (info@brockmann-consult.de)

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```